



下载APP



03 | 初窥门径：开发并运行你的第一个 eBPF 程序

2022-01-21 倪朋飞

《eBPF 核心技术与实战》

课程介绍 >

**讲述：山荣**

时长 13:27 大小 12.33M



你好，我是倪朋飞。

通过前面两讲，我已经带你为正式进入 eBPF 的学习做好了准备，接下来我们进入第二个模块“基础入门篇”的学习。在这个模块里，你会学习到 eBPF 的开发环境搭建方法、运行原理、编程接口，以及各种类型 eBPF 程序的事件触发机制和应用场景。

上一讲，我带你一起梳理了 eBPF 的学习路径和学习技巧，并特别强调了动手实践在学习 eBPF 过程中的重要性。那么，eBPF 程序到底是什么样子的？如何搭建 eBPF 的开发环境，又如何开发一个 eBPF 应用呢？



今天，我就带你一起上手开发第一个 eBPF 程序。

如何选择 eBPF 开发环境？

在前两讲中，我曾提到，虽然 Linux 内核很早就已经支持了 eBPF，但很多新特性都是 [在 4.x 版本中逐步增加的](#)。所以，想要稳定运行 eBPF 程序，内核至少需要 **4.9** 或者更新的版本。而在开发和学习 eBPF 时，为了体验和掌握最新的 eBPF 特性，我推荐使用更新的 **5.x** 内核。

作为 eBPF 最重大的改进之一，一次编译到处执行（简称 CO-RE）解决了内核数据结构在不同版本差异导致的兼容性问题。不过，在使用 CO-RE 之前，内核需要开启 `CONFIG_DEBUG_INFO_BTTF=y` 和 `CONFIG_DEBUG_INFO=y` 这两个编译选项。为了避免你在首次学习 eBPF 时就去重新编译内核，我推荐使用已经默认开启这些编译选项的发行版，作为你的开发环境，比如：

Ubuntu 20.10+

Fedora 31+

RHEL 8.2+

Debian 11+

你可以到公有云平台上创建这些发行版的虚拟机，也可以借助 [Vagrant](#)、[Multipass](#) 等工具，创建本地的虚拟机。比如，使用我最喜欢的 [Vagrant](#)，通过下面几步就可以创建一个 Ubuntu 21.10 的虚拟机：

 复制代码

```
1 # 创建和启动Ubuntu 21.10虚拟机
2 vagrant init ubuntu/impish64
3 vagrant up
4
5 # 登录到虚拟机
6 vagrant ssh
```

如何搭建 eBPF 开发环境？

虚拟机创建好之后，接下来就需要安装 eBPF 开发和运行所需要的开发工具，这包括：

将 eBPF 程序编译成字节码的 LLVM；

C 语言程序编译工具 make ；

最流行的 eBPF 工具集 BCC 和它依赖的内核头文件 ；

与内核代码仓库实时同步的 libbpf ；

同样是内核代码提供的 eBPF 程序管理工具 bpftool。

你可以执行下面的命令，来安装这些必要的开发工具：

 复制代码

```
1 # For Ubuntu20.10+
2 sudo apt-get install -y make clang llvm libelf-dev libbpf-dev bpfcc-tools lib
3
4 # For RHEL8.2+
5 sudo yum install libbpf-devel make clang llvm elfutils-libelf-devel bpftool bc
```

如果你已经熟悉了 Linux 内核的自定义编译和安装方法，并选择在旧的发行版中通过自行编译和升级内核搭建开发环境，上述的开发工具流程也需要做适当的调整。这里特别提醒下，libbpf-dev 这个库很可能需要从源码安装，具体的步骤你可以参考 libbpf 的 [GitHub 仓库](#)。

如何开发第一个 eBPF 程序？

当前面这些开发工具和依赖库安装完成后，一个完整的 eBPF 开发环境就准备好了。接下来，你肯定迫不及待地想要体验一下 eBPF 的强大功能了。

不过，在开发 eBPF 程序之前，我们先来看一下 eBPF 的开发和执行过程。如下图（图片来自 [brendangregg.com](#)）所示，一般来说，这个过程分为以下 5 步：

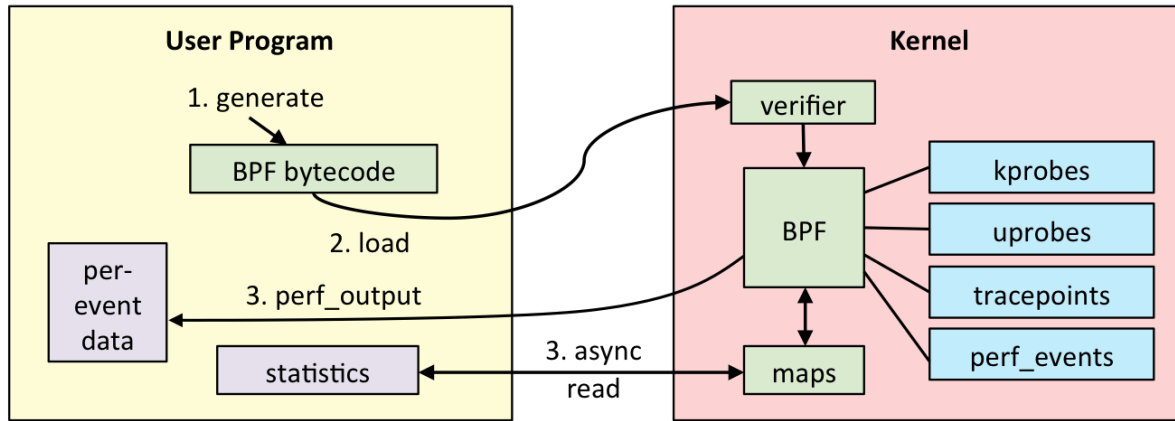
第一步，使用 C 语言开发一个 eBPF 程序；

第二步，借助 LLVM 把 eBPF 程序编译成 BPF 字节码；

第三步，通过 bpf 系统调用，把 BPF 字节码提交给内核；

第四步，内核验证并运行 BPF 字节码，并把相应的状态保存到 BPF 映射中；

第五步，用户程序通过 BPF 映射查询 BPF 字节码的运行状态。



eBPF 程序执行过程

这里的每一步，我们当然可以自己动手去完成。但对初学者来说，我推荐你从 BCC (BPF Compiler Collection) 开始学起。

BCC 是一个 BPF 编译器集合，包含了用于构建 BPF 程序的编程框架和库，并提供了大量可以直接使用的工具。使用 BCC 的好处是，**它把上述的 eBPF 执行过程通过内置框架抽象了起来，并提供了 Python、C++ 等编程语言接口**。这样，你就可以直接通过 Python 语言去跟 eBPF 的各种事件和数据进行交互。

接下来，我就以跟踪 `openat()`（即打开文件）这个系统调用为例，带你来看看如何开发并运行第一个 eBPF 程序。

使用 BCC 开发 eBPF 程序，可以把前面讲到的五步简化为下面的三步。

第一步：使用 C 开发一个 eBPF 程序

新建一个 `hello.c` 文件，并输入下面的内容：

```
1 int hello_world(void *ctx)
2 {
3     bpf_trace_printk("Hello, World!");
4     return 0;
5 }
```

[复制代码](#)

就像所有编程语言的 “Hello World” 示例一样，这段代码的含义就是打印一句 “Hello, World!” 字符串。其中，`bpf_trace_printk()` 是一个最常用的 BPF 辅助函数，它的作用是输出一段字符串。不过，由于 eBPF 运行在内核中，它的输出并不是通常的标准输出（`stdout`），而是内核调试文件 `/sys/kernel/debug/tracing/trace_pipe`，你可以直接使用 `cat` 命令来查看这个文件的内容。

第二步：使用 Python 和 BCC 库开发一个用户态程序

接下来，创建一个 `hello.py` 文件，并输入下面的内容：

 复制代码

```
1 #!/usr/bin/env python3
2 # 1) import bcc library
3 from bcc import BPF
4
5 # 2) load BPF program
6 b = BPF(src_file="hello.c")
7 # 3) attach kprobe
8 b.attach_kprobe(event="do_sys_openat2", fn_name="hello_world")
9 # 4) read and print /sys/kernel/debug/tracing/trace_pipe
10 b.trace_print()
```

让我们来看看每一处的具体含义：

第 1) 处导入了 BCC 库的 BPF 模块，以便接下来调用；

第 2) 处调用 `BPF()` 加载第一步开发的 BPF 源代码；

第 3) 处将 BPF 程序挂载到内核探针（简称 `kprobe`），其中 `do_sys_openat2()` 是系统调用 `openat()` 在内核中的实现；

第 4) 处则是读取内核调试文件 `/sys/kernel/debug/tracing/trace_pipe` 的内容，并打印到标准输出中。

在运行的时候，BCC 会调用 LLVM，把 BPF 源代码编译为字节码，再加载到内核中运行。

第三步：执行 eBPF 程序

用户态程序开发完成之后，最后一步就是执行它了。需要注意的是，eBPF 程序需要以 `root` 用户来运行，非 `root` 用户需要加上 `sudo` 来执行：

 复制代码

```
1 $ chmod +x hello.py
2 $ sudo ./hello.py
```

稍等一会，你就可以看到如下的输出：

 复制代码

```
1 b' cat-10656 [006] d... 2348.114455: bpf_trace_printk: Hello, World!'
```

输出的格式可由 `/sys/kernel/debug/tracing/trace_options` 来修改。比如前面这个默认的输出中，每个字段的含义如下所示：

`cat-10656` 表示进程的名字和 PID；

`[006]` 表示 CPU 编号；

`d...` 表示一系列的选项；

`2348.114455` 表示时间戳；

`bpf_trace_printk` 表示函数名；

最后的 `"Hello, World!"` 就是调用 `bpf_trace_printk()` 传入的字符串。

到了这里，恭喜你已成功开发并运行了第一个 eBPF 程序！不过，短暂的兴奋之后，你会发现这个程序还有不少的缺点，比如：

既然跟踪的是打开文件的系统调用，除了调用这个接口进程的名字之外，被打开的文件名也应该在输出中；

`bpf_trace_printk()` 的输出格式不够灵活，像是 CPU 编号、`bpf_trace_printk` 函数名等内容没必要输出；

.....

实际上，我并不推荐通过内核调试文件系统输出日志的方式。一方面，它会带来很大的性能问题；另一方面，所有的 eBPF 程序都会把内容输出到同一个位置，很难根据 eBPF 程序去区分日志的来源。

那么，怎么来解决这些问题呢？接下来，我们就试着一起改进这个程序。

如何改进第一个 eBPF 程序？

在 [01 讲](#) 中我曾提到，BPF 程序可以利用 BPF 映射（map）进行数据存储，而用户程序也需要通过 BPF 映射，同运行在内核中的 BPF 程序进行交互。所以，为了解决上面提到的第一个问题，即获取被打开文件名的问题，我们就要引入 BPF 映射。

为了简化 BPF 映射的交互，BCC 定义了一系列的 [库函数和辅助宏定义](#)。比如，你可以使用 BPF_PERF_OUTPUT 来定义一个 Perf 事件类型的 BPF 映射，代码如下：

[复制代码](#)

```
1 // 包含头文件
2 #include <uapi/linux/openat2.h>
3 #include <linux/sched.h>
4
5 // 定义数据结构
6 struct data_t {
7     u32 pid;
8     u64 ts;
9     char comm[TASK_COMM_LEN];
10    char fname[NAME_MAX];
11 };
12
13 // 定义性能事件映射
14 BPF_PERF_OUTPUT(events);
```

然后，在 eBPF 程序中，填充这个数据结构，并调用 perf_submit() 把数据提交到刚才定义的 BPF 映射中：

[复制代码](#)

```
1 // 定义kprobe处理函数
2 int hello_world(struct pt_regs *ctx, int dfd, const char __user * filename, st
3 {
4     struct data_t data = { };
5
6     // 获取PID和时间
7     data.pid = bpf_get_current_pid_tgid();
8     data.ts = bpf_ktime_get_ns();
9
10    // 获取进程名
11    if (bpf_get_current_comm(&data.comm, sizeof(data.comm)) == 0)
```

```
12  {
13      bpf_probe_read(&data.fname, sizeof(data.fname), (void *)filename);
14  }
15
16  // 提交性能事件
17  events.perf_submit(ctx, &data, sizeof(data));
18  return 0;
19 }
```

其中，以 bpf 开头的函数都是 eBPF 提供的辅助函数，比如：

bpf_get_current_pid_tgid 用于获取进程的 TGID 和 PID。因为这儿定义的 data.pid 数据类型为 u32，所以高 32 位舍弃掉后就是进程的 PID；

bpf_ktime_get_ns 用于获取系统自启动以来的时间，单位是纳秒；

bpf_get_current_comm 用于获取进程名，并把进程名复制到预定义的缓冲区中；

bpf_probe_read 用于从指定指针处读取固定大小的数据，这里则用于读取进程打开的文件名。

有了 BPF 映射之后，前面我们调用的 bpf_trace_printk() 其实就不再需要了，因为用户态进程可以直接从 BPF 映射中读取内核 eBPF 程序的运行状态。

这其实也就是上面提到的第二个待解决问题。那么，怎样从用户态读取 BPF 映射内容并输出到标准输出 (stdout) 呢？

在 BCC 中，与 eBPF 程序中 BPF_PERF_OUTPUT 相对应的用户态辅助函数是 open_perf_buffer()。它需要传入一个回调函数，用于处理从 Perf 事件类型的 BPF 映射中读取到的数据。具体的使用方法如下所示：

```
1  from bcc import BPF
2
3  # 1) load BPF program
4  b = BPF(src_file="trace-open.c")
5  b.attach_kprobe(event="do_sys_openat2", fn_name="hello_world")
6
7  # 2) print header
8  print("%-18s %-16s %-6s %-16s" % ("TIME(s)", "COMM", "PID", "FILE"))
9
10 # 3) define the callback for perf event
```

 复制代码

```
11 start = 0
12 def print_event(cpu, data, size):
13     global start
14     event = b["events"].event(data)
15     if start == 0:
16         start = event.ts
17     time_s = (float(event.ts - start)) / 1000000000
18     print("%-18.9f %-16s %-6d %-16s" % (time_s, event.comm, event.pid, event.f
19
20 # 4) loop with callback to print_event
21 b["events"].open_perf_buffer(print_event)
22 while 1:
23     try:
24         b.perf_buffer_poll()
25     except KeyboardInterrupt:
26         exit()
```

让我们来看看每一处的具体含义：

第 1) 处跟前面的 Hello World 一样，加载 eBPF 程序并挂载到内核探针上；

第 2) 处则是输出一行 Header 字符串表示数据的格式；

第 3) 处的 `print_event` 定义一个数据处理的回调函数，打印进程的名字、PID 以及它调用 `openat` 时打开的文件；

第 4) 处的 `open_perf_buffer` 定义了名为 “events” 的 Perf 事件映射，而后通过一个循环调用 `perf_buffer_poll` 读取映射的内容，并执行回调函数输出进程信息。

将前面的 eBPF 程序保存到 `trace-open.c`，然后再把上述的 Python 程序保存到 `trace-open.py` 之后（你可以在 GitHub [@ebpf-apps](#) 上找到完整的代码），就能以 `root` 用户来运行了：

 复制代码

```
1 chmod +x trace-open.py
2 sudo ./trace-open.py
```

稍等一会，你会看到类似下面的输出：

	TIME(s)	COMM	PID	FILE	
1	2.384485400	b'irqbalance'	991	b'/proc/interrupts'	 复制代码
2	2.384750400	b'irqbalance'	991	b'/proc/stat'	
3	2.384838400	b'irqbalance'	991	b'/proc/irq/0/smp_affinity'	
4					

恭喜，你已经开发了第一个完整的 eBPF 程序。相对于前面的 Hello World，它的输出不仅格式更为清晰，还把进程打开的文件名输出出来了，这在排查频繁打开文件相关的性能问题时尤其有用。

小结

今天，我带你一起搭建了 eBPF 的开发环境，安装了 eBPF 开发时常用的工具和依赖库。并且，我从最简单的 Hello World 开始，带你借助 BCC 从零开发了一个跟踪 [openat\(\)](#) 系统调用的 eBPF 程序。

通常，开发一个 eBPF 程序需要经过开发 C 语言 eBPF 程序、编译为 BPF 字节码、加载 BPF 字节码到内核、内核验证并运行 BPF 字节码，以及用户程序读取 BPF 映射五个步骤。使用 BCC 的好处是，它把这几个步骤通过内置框架抽象了起来，并提供了简单易用的 Python 接口，这可以帮你大大简化 eBPF 程序的开发。

除此之外，BCC 提供的一系列工具不仅可以直接用在生产环境中，还是你学习和开发新的 eBPF 程序的最佳参考示例。在课程后续的内容中，我还会带你深入 BCC 的详细使用方法。

思考题

最后，我想请你聊一聊这几个问题：

1. 你通常都是如何搭建 Linux 和 eBPF 环境的？
2. 在今天的案例操作中，你遇到了什么问题，又是如何解决的呢？
3. 虽然今天开发的程序非常短小，你觉得它能否在日常的工作中帮助到你呢？

欢迎在留言区和我讨论，也欢迎把这节课分享给你的同事、朋友。让我们一起在实战中演练，在交流中进步。

分享给需要的人，Ta 订阅超级会员，你将得 50 元

Ta 单独订阅本课程，你将得 20 元

生成海报并分享

赞 4 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 02 | 先利其器：如何高效学习 eBPF？

精选留言 (7)

写留言



莫名 置顶

2022-01-21

追踪文件打开事件，采用场景大致有：

- 1、查看某个程序启动时加载了哪些配置文件，便于确认是否加载了正确的配置文件。对于允许自定义配置文件路径的程序尤其有用，例如 MySQL、PostgreSQL。
- 2、查看是否存在频繁或周期性打开某些文件的情况，考虑是否存在优化可能。比如周期性打开某个极少变化的文件，可以一次性读取，且监听文件变动事件，避免多次打开读取...

展开

作者回复: 太赞了 感谢分享追踪文件打开的这些应用场景



6



莫名 置顶

2022-01-21

给倪老师的 Github 仓库里面的这个小程序挑个 bug：

- 1、`b = BPF(src_file="trace-open.c")`，`trace-open.c -> trace_open.c`
- 2、依赖了 `openat2.h` 头文件，`openat2` 系统调用自 5.6 版本才出现，低于这个版本无法运行 `trace_open.py`，建议用 `openat` 系统调用，有比较好的版本兼容性。

展开

作者回复: 1. 谢谢指出，已经修复了。

2. 是的，但我们专栏还是希望使用新版本的内核来学习，这样一方面能体验最新的特性，另一方

面不需要等后面需要用到新特性的时候再重新配置开发环境。



1

**Geek_5aa343**

2022-01-21

老师，您好，有几个问题请教下：

1. "do_sys_openat2() 是系统调用 openat() 在内核中的实现" 怎么去找到一个系统调用在内核中的实现呢？
2. 使用BPF map获取openat的打开文件名这里，// 定义数据结构struct data_t { u32 pid; u64 ts; char comm[TASK_COMM_LEN]; char fname[NAME_MAX];}; 这个的格式为啥是这...
展开

作者回复: 1. 每个系统调用在内核中都有类似 sys_xxx 的实现函数，我们后面课程会讲到怎么查询内核中的跟踪点，根据名字去过滤就可以找到实现函数的名字。当然，如果你要看具体的实现步骤，就得需要去看看内核的源码了。

2. 没有固定的格式，这个数据结构其实就是你想在用户态中获取的数据，不同的程序需要的数据是不同的。



1

**Captain perison**

2022-01-21

老师，你好，在arm嵌入式环境中，有没有办法添加这些依赖的库？

作者回复: 不好意思，我不太确定。你的内核版本和发行版用的是啥？我看Andriod文档上面是可以用 eBPF 的 <https://source.android.com/devices/architecture/kernel/bpf?hl=zh-cn>



1

**晴天**

2022-01-21

老师，请问下生产环境使用ebpf的话，哪个版本的内核合适啊？

作者回复: 为了确保生产环境的稳定性，一般推荐让ebpf程序适配内核版本，而不是反过来为了ebpf而去升级内核。当然，新版本内核有些特性也是没法在旧版本中使用的，还要根据实际需要去看。

BCC的文档里面列出了新特性对应的内核版本，可作参考 <https://github.com/iovisor/bcc/blob/master/docs/kernel-versions.md>



ThinkerWalker

2022-01-21

有个Python语法的疑问：trace_open.py中，print_event被调用的时候【28行，b["events"].open_perf_buffer(print_event)】是如何传入三个参数（cpu, data, size）呢？

作者回复：这些参数都是由BCC框架默认提供的，不需要使用的时候再额外传入。



草根

2022-01-21

请问老师，hello_world的入参是要和系统调用的入参设置一致吗？

作者回复：这个示例实际上是kprobe类型的eBPF程序，它的参数格式是表示寄存器上下文的struct pt_regs *ctx和内核函数的实际参数。后面的课程还会详细介绍eBPF支持的各种程序类型。

